CS501
VOTE: Visual Object Tracing Engine

The Internal Documentation

Carrer, Marco: mc120@cornell.edu
Chang, Thomas: hc75@cornell.edu
Lin, Paul: pil3@cornell.edu
Stein, Chris: cas38@cornell.edu

December, 5 1997

Table of Contents

Some Information **4**

System Architecture 5

Subsystem Diagram: Preprocessor 6

The Preprocessing Engine **6**

Tcl/Tk and C++ 8

Preprocessor Flow Chart 9

Order of Events in the Preprocessor **9**

cl_Preprocessor 11

Inheritance **11**

Interfaces **11**

Internals **11**

Configuration Interpreter 13

cl_PrefSet 14

Interfaces **14**

Internals **14**

cl_RunSet 16

Interfaces **16**

Internals **16**

cl_ClassHash 17

Interfaces **17**

Internals **17**

Log File 18

cl_Log 19

Interface **19**

Internals **20**

cl_TimeLine & cl_LogEvent 21

cl_TimeLine Interface **21**

cl_LogEvent Interface **21**

cl_StringTokenizer 23

Interface **23**

Internals **23**

cl_LineFileStream 24
Interface **24**
Rendering Engine: cl_MSCDrawer 25
Internals **26**
Interfaces **27**
Limitations and Assumption **27**
cl_Gui 28
Internals **28**
Interfaces **28**
Subsystem Diagram: Main GUI 29
The Main GUI **29**
Main GUI Flow Chart 31
Order of Events in the Main GUI **31**
Limitations and Assumptions 32
Testing 33
Individual Subsystems **33**
Merged System **33**

Some Information

Status of Project: Tested, documented, and **ready for download** from

http://www.csuglab.cornell.edu/Info/People/hc75/initSpec.htm

Technologies Used:

Tcl/Tk, release 8.0

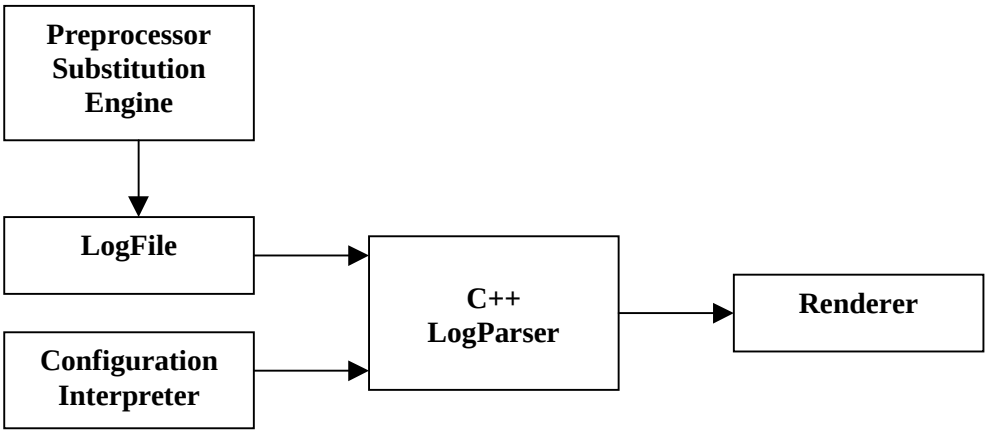
C++, with Microsoft Visual C++ 5.0.

Purify, to check for memory leaks.

Lines of Code:

C++ code	4,800
Tcl/Tk	3,000

System Architecture

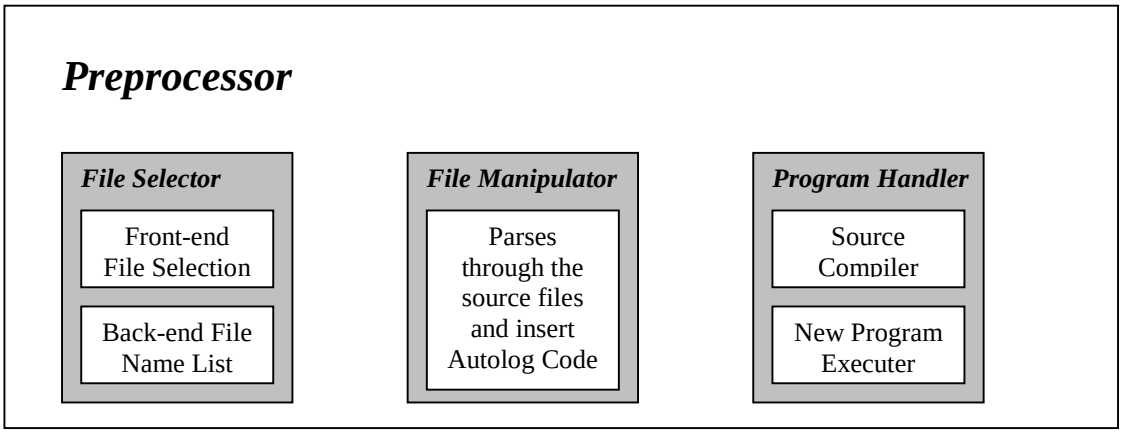


As highlighted in the first project proposal, the software architecture of VOTE consists of five subsystems. The above diagram depicts the subsystems and their high-level interconnections.

1. Preprocessor and Substitution Engine: This collects the names of all the methods declared in a class. We use the regular expression functionality of Tcl in our approach to parse method declarations. To do this we call the Tcl Interpreter within our C/C++ code. The preprocessor also parses and replaces the method declaration line, inserting an AutoLog object declaration immediately afterward in the body of the routine.
2. Configuration Interpreter: Sets up the configurations of the tool so that it "knows" how to rebuild the project and run it. This is done through a GUI dialog box, where the user is asked for the commands to accomplish this. We also require the command to run the project, because the log file is produced at run time.
3. Log File: A log-file format has been defined. This log-file is written to by AutoLog objects as the system executes.
4. Log File Parser: This parser reads in the log-file and synthesizes the log-file information into a series of events. This information is made available to the MSC renderer through a simple iterator interface.
5. Rendering Engine: The rendering engine queries the log-file parser for event information and draws with the canvas widget of Tcl/Tk.
6. In addition, our system has a GUI subsystem.

In the following sections we offer a detailed introduction to each of the modules presented above. For each section, we offer some architecture diagrams, class diagrams, and flow diagrams. The interfaces supported of key classes are also introduced.

Subsystem Diagram: Preprocessor



The Preprocessing Engine

The front end of the Preprocessor is implemented in Tcl/Tk and the backend is done in C++. This subsystem of VOTE handles the preparation of relevant data before the actual analysis, which will in the end generate the MSC output. This step goes from the parsing, transforming, of source files, to the compilation and execution of the program. In the process it cooperates with the PrefSet module that keeps track of all the user-selected preferences. The word 'preprocessor' is a something of a misnomer because it doesn't preprocess source files in the conventional use of the word. Instead it generates log files for the drawing engine of VOTE.

File Selector

The File Selector Module is the first part that is implemented in Tcl/Tk. It is responsible for letting the user specify which files he wishes us to act on. For example, the user can point to a directory and specify a file filter to insert files into a storage place for the names. The purpose of this is to have a default set of file names for every run (draw) of our tool. The user can add or remove file names from this list with button clicks. This is implemented using two list boxes, one specifying files in the directory and one specifying the cumulative list of files the user has chosen. There are two buttons in the middle to facilitate the user in moving files from one list box to the other. This will be described further in the user manual.

File Manipulator

The File Manipulator Module is the only part of this subsystem that is implemented in C++. The class responsible for this is cl_Preprocessor. It scans each of the files that the File Selector gives it, for all the function definitions, saves this information, and insert snippets of code in each implementation to "trace" function calls. The code that is inserted is just an Autolog object declaration. For purposes of not conflicting with the code already there, we have chosen the class name "_mc120_hc75_pil3_cas38_cl_Autolog". The concept behind the AutoLog code insertion, as described in our description of the project, is to statically construct an AutoLog object in the beginning of the function, which will automatically log the entrance into the function. Leaving of the function will also be logged by the destructor of the AutoLog class as the object goes out of scope. In addition to the snippets of code, the File Manipulator also inserts a line in the beginning

of the file <#include "_mc120_hc75_pil3_cas38_cl_Autolog.h">. This file contains inline methods that handle all the logging.

Program Handler

The Program Handler Module is responsible for compiling the altered source code that the File Manipulator Module outputted as well as running the newly compiled program. This can all be done easily with a script. Therefore, it is done in Tcl/Tk. We need, however, information from the PrefSet module, like build options, to be able to have a functional compile/run phase. Attaining this information is also through Tcl/Tk using a set of global variables since the PrefSet module also plugs in the Tcl/Tk script.

Tcl/Tk and C++

Since the project is done in both Tcl/Tk and C++, we should discuss how the two languages work together. The intuition is to have the Tcl script generate the main interface with the user, providing the GUI, and have the C++ modules do all the dirty work. The two main modules that make up VOTE are the vote.exe executable and the vote.tcl Tcl/Tk script. The main mode of interaction is done through calls to Tcl_CreateCommand in the main() function. Every call to Tcl_CreateCommand will set up a link between a static method of some class and a "proc" name that will be used in the script. The word script really refers to the main vote.tcl and all the other Tcl files that it sources. The following are descriptions of such interactions grouped by modules.

The Preprocessor

As mentioned before, the preprocessor is both in Tcl/Tk and C++. Tcl./Tk handles the front end while C++ takes care of the back end. The main entry point for the work of the Preprocessor is through a static member function of the aforementioned cl_Preprocessor class, called DoFile. This instantiates a new cl_Preprocessor and calls its Do function on the selected file.

The PrefSet Module

The PrefSet class offers two method SetAllPref() and GetAllPref() for the Tcl/Tk script to update and acquire the preferences parameters. In the beginning of vote.tcl, GetAllPref() is invoked to load the global parameter variables in C++ module into global variables in Tcl/Tk script. After the user makes changes to the parameters in the preference window and 'OK' button is clicked, SetAllPref() is invoked to transport the new settings back to the C++ module.

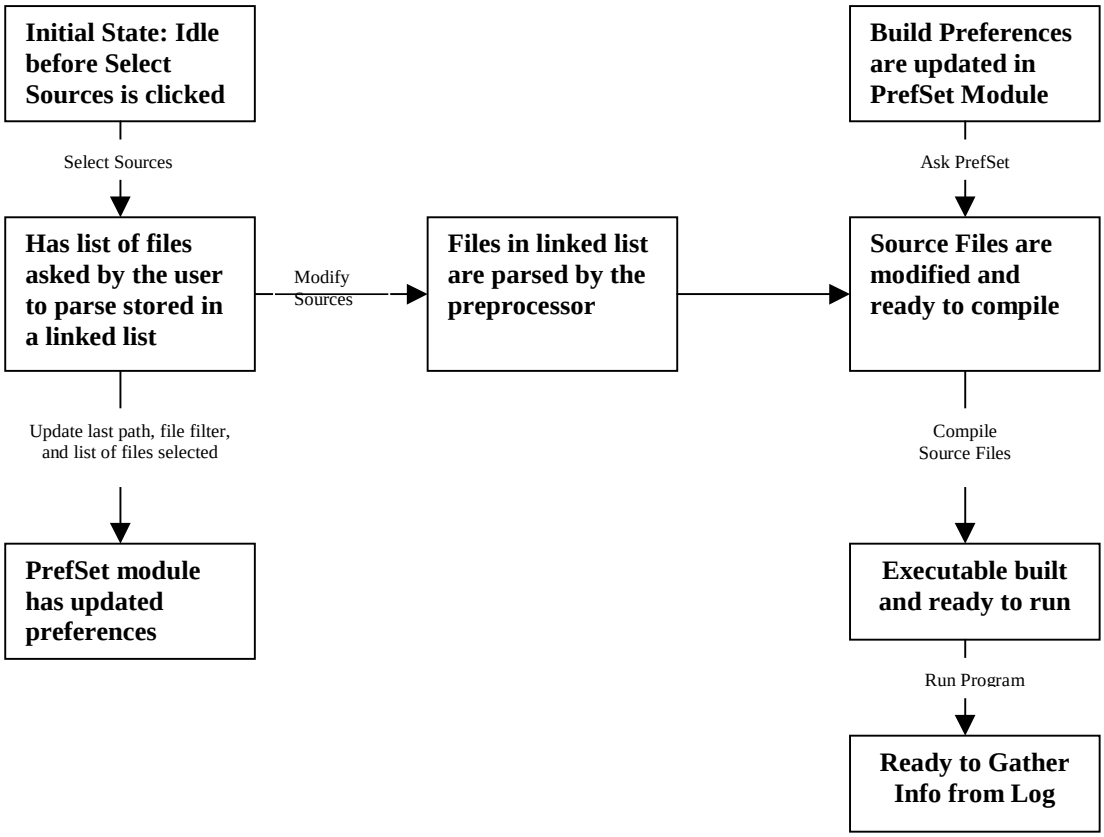
The RunSet Module

The available and viewable classes and methods are kept in two hash table structures in C++ module. The RunSet class provides two methods, *GetClassList* and *GetMethodList* to acquire information on all available classes. Access to the viewable classes are achieved by a series of public methods, which are: *FilterInsertClass, FilterInsertMethod, FilterDeleteClass, FilterDeleteMethod*. All the actions are invoked thorough a command dispatcher *DispatchRunSet*.

The MSC Module

The MSC module takes the information analyzed from the log file and generates a Tcl/Tk script file that will be run. This choice was taken in that we will use Tcl/Tk for our GUI, which naturally includes the final output.

Preprocessor Flow Chart



Order of Events in the Preprocessor

The main menu button that is associated with the Preprocessor is the Build menu button. The following steps all involve the user's interaction with menu items under this.

The user first clicks on *Select Sources* to select what files he wants VOTE to parse and modify, and use to draw an MSC. The last directory scanned by the user is saved and exported to the PrefSet module for future default settings. The set of files that the user chooses marks the first filtering stage of VOTE. As this is less frequent, the information is more static and therefore persistent; we dump this information to a file along with other fields of the preferences. This is also covered in the description of the PrefSet class.

At the request of the user by clicking on the *Modify Sources* menu item, the core of the Preprocessor kicks in. Using the class cl_Preprocessor, the files selected in Tcl/Tk are modified by the preprocessor through a static method that this class exports, called **DoFile**. The modifications made to the files as described earlier will enable the logging functionality while executing the program later.

So now that the files are ready to compile, we need to rely on the user's preferences on what to execute when building his project. Unfortunately there are certain limitations that we must adhere to. All these will be further elaborated in the Limitations and Assumptions. The user specifies

this as part of the general preferences. When he clicks on the *Build* menu item, the building processes initiates using the user's preferences for compiler/linker commands and compiling directory. The status box at the bottom of the window will let the user know when building is done.

Finally the user will click on the *Run* menu item that initiates the execution of the altered program. This will dump logging information in a file with a preset name. According to the user's preferences this file will be copied to the name that he specified.

cl_Preprocessor

The class cl_Preprocessor is the core of the preprocessing engine. It "plugs in" to the Tcl/Tk script like the other modules and uses information provided by the other classes like the cl_PrefSet class. The main functionality that this class exports is the *DoFile* function. It is the entry point of access to and from the Tcl/Tk script. The link (between C++ and Tcl/Tk) is set up in the main() function in main.cpp. The function *DoFile* parses files and inserts snippets of code into them, specifically at the beginning of every function to log the entrance and exiting of each function. As the requirements of Tcl./Tk indicate, this method is made static. It instantiates an instance of the cl_Preprocessor class and calls an internal function *Do* for each file.

Inheritance

This class inherits from the cl_Gui class because via that class, error messages can be directed to message boxes in Tcl/Tk. Other classes such as cl_MSCDrawer and cl_RunSet all inherit from this class for the same reason. To have it function correctly we need to add some lines to set up a link inside the static method *DoFile*.

Interfaces

```
static DoFile(char* szFileName);
```

As mentioned above, this is what the main() function links a Tcl procedure called *DoFile* (same name to keep things clear) to. In the Tcl script, we can then invoke this function by simply calling *DoFile*. This function, in turn, calls an internal function called *Do* after checking that the file has not been modified.

Internals

Private Data Members

Private member objects used to store all the locations of where to look for function definitions so the modifier can insert code into the functions.

A helper class used is the cl_IndexPair that denotes a pair of indices in the string (what the file has been read into) where the function name is likely to appear.

Another data structure that was useful was the cl_Entry that stores a class name with a method name that belongs to the class. This may introduce duplicates in the class names. This is a concern only from a memory space minimization standpoint – not one of our key design concerns.

The private data fields are:

```
cl_Entry m_eTable[MAX_NUM_FUNCS];
```

This stores each entry that was parsed, i.e. each function that was read by the parser.

```
cl_IndexPair m_ipTable[MAX_NUM_FUNCS];
```

This stores each "}" "{" pair that was parsed. This is useful when looking for functions because the function name of a function will always be between a close braces and an open braces.

```
int m_iNumEntries;
```

```
int m_iNumIndexPairs;
```

This keeps track of how many of which are in the arrays.

Private Methods

Most functions of this class are kept private because we only export one function. The list of private methods of this class includes the following.

1) This is called by DoFile to really modify the file with the name szFileName. It first does a backup and then does the parsing/inserting.

```
Error Do(char* szFileName, char* szBackup);
```

2) This method parses the file and records where all the open braces and close braces are to set the stage for the next step involving parsing/inserting.

```
Error ReadIndexPairs(char* szBuf);
```

3) This method searches for the class name given a starting index (where the :: is). Thus the search is backwards.

```
Error FindClassName(char* szBuf, char* szName, int iStart);
```

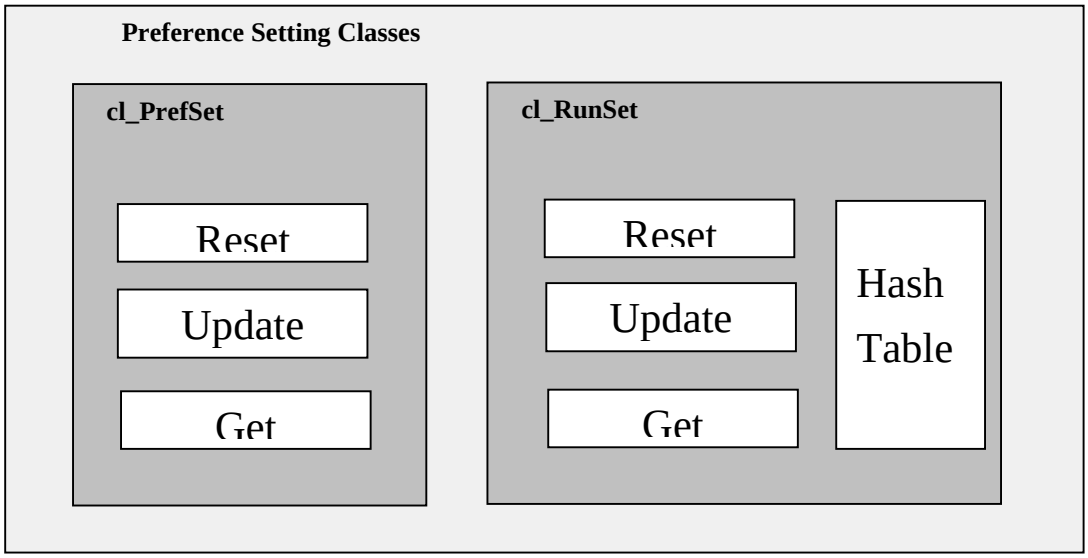
4) This method searches for the method name, plus the signature of the method, also given a starting index (where the :: is). Unlike the class name searcher, this, being the one to search after the class name has been found, returns an indicator of where it was found. This is in the form of an index into the m_ipTable and will be stored together with the class name and method name of the current entry.

```
Error FindMethodName(char* szBuf, char* szName,
                     int iStart, int& iIndex);
```

5) This writes the modified string back to the file. The insertion is taken care of easily because while inserting, we check if we hit an index where we need to insert by looking into the m_eTable; remember that an entry in m_eTable stores an index to a corresponding m_ipTable entry. At those indices, we insert not only the braces themselves, but also the snippets of code to enable logging.

```
Error WriteToFile(ofstream& ofstr, char* szBuf);
```

Configuration Interpreter

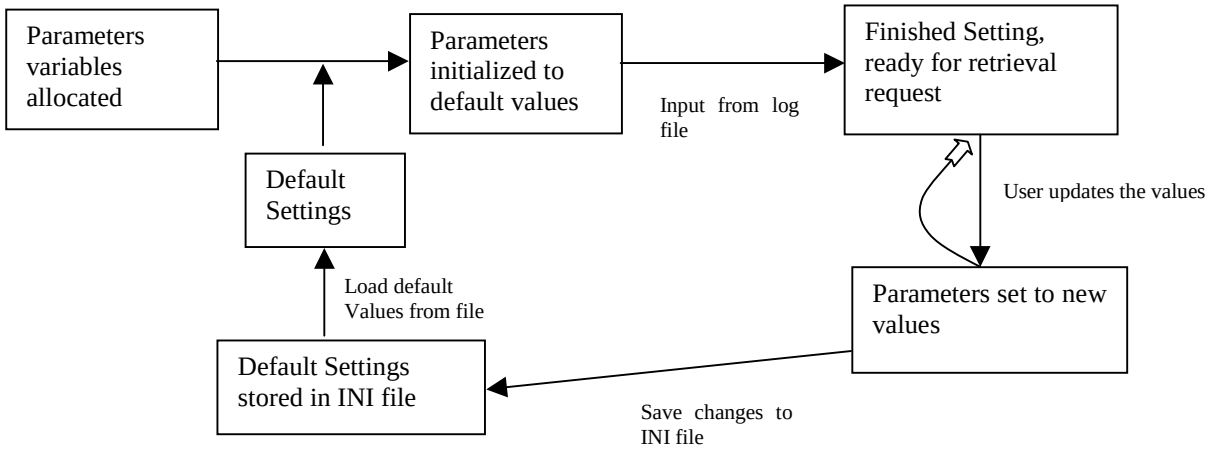


There are two sets of configuration parameters that specify the run-time preferences controlling all the available functionalities. There are two classes which provide the data storage all the parameters in each set and offer public access of the data.

The cl_PrefSet contains the environmental settings that users specify at the beginning and do not change. For example, the user is allowed to specify the type of compiler and linker to generate executables. File extension (*.cc, *.cpp or *.c) of the source code is also specified at the beginning. Parameters that users adjust regularly, such as the visible classes in the MSC, are stored in cl_RunSet.

Each class consists of three components; Reset, Set and Get. Reset is used for setting all the values back to the default ones stored in the default objects. Set and Get are responsible for accessing the data. In RunSet, a hash table class is needed to maintain a list of class information.

Flow Chart



cl_PrefSet

Cl_PrefSet is responsible for maintaining the preference parameters specified by the user. A group of private member static data fields are used for storing the data. Three major public methods will be provided for other classes to access the parameters. Two methods are created to transport data member variables to and from the Tcl/Tk script

The data member variables and public methods are defined as static. The cl_PrefSet class is never instantiated by other classes. Access to the preference parameters is achieved by invoking the static member methods.

Interfaces

```
cl_PrefSet::SetPar(FieldName, char* Value);
```

Other classes can use this function to update any setting field. Field is an enumerated data type with the names of all the fields in cl_PrefSet.

```
cl_PrefSet::GetPar(FieldName, char* Value);
```

This method is used for retrieving the current setting of preferences. Field and *FieldType* are described above.

```
static int SetAllPar(ClientData Data, Tcl_Interp* interp, int argc, char* argv[]);
```

```
static int GetAllPar(ClientData Data, Tcl_Interp* interp, int argc, char* argv[]);
```

These two methods are invoked in Tcl/Tk script to acquire or modify the preference parameters. The correspondence between the order of arguments from Tcl/Tk script and that of global variables in C++ code is coordinated by the enum type *FieldNames.*

```
static Error PrefLoadFile();
```

```
static Error PrefSaveFile();
```

The two functions are for loading and saving preferences parameters on disk to preserve persistent setting in every execution. PrefLoadFile() is automatically invoked at the beginning of execution and PrefSaveFile() is called when user clicked the “OK” button in the preference window in GUI.

Internals

Private member objects exist to contain all the preference settings. All the public member functions are to access the private data members

The private data fields are:

`char fileExt:` possible file extensions

`char LastPath:` the last path where the user include the source files

`char compilerCommand:` application executable name of the compiler to use and arguments

`char linkerCommand:` application executable name of the linker to use

`char logFileName:` file name of the log file

`char execFileName:` file name of the log file

`char compileDir:` compile directory for the executable

`char bgColor:` Background Color, RGB values, for example black = "000000", white="ffffff"

`char bxColor:` Box Color

`char retArrowColor:` return arrow color

`char entArrowColor:` enter arrow color

`char timeLineColor:` time line color

`char textColor:` text color

`char xSpacing:` space in pixels between 2 time lines

`char ySpacing:` space in pixels between two adjacent events

The first time when VOTE is executed, PrefLoadFile() will look for the default initialization file “default.ini” in the working directory. If the user modified the preference setting, a new initialization file “user.ini” will be saved. Subsequent executions will look for the file “user.ini” first. If it is not found, “default.ini” will be used instead. If both are not found in the current working directory, execution will be terminated and error reported since VOTE needs preference parameters which specify the behavior of most actions. PrefSaveFile() will overwrite “user.ini” every time.

cl_RunSet

During execution after extracting information from the log file, all the available classes and the methods will be identified. Two lists will be kept to track this information. The first one list keeps track of all the existing classes and methods in the log file. The user has the freedom to choose which classes and methods to be shown in the MSC diagram. The second list holds the selections

The lists, implemented using hash tables, are maintained by this class (cl_RunSet). Two instances of cl_ClassHash, sm_ClassHash and sm_FilterHash, are kept as static private members. (Class cl_ClassHash will be discussed in detail in a later section.) This class provides public access to the contents of above two lists.

Due to the fact that cl_RunSet will interact closely with Tcl/Tk script, a dispatcher is created to invoke all the class methods to avoid passing multiple interpreter-dependent variables to each member method.

Interfaces

`DispatchRunSet(ClientData Data, Tcl_Interp* interp, int argc, char* argv[]):` TCL/TK command dispatcher

`LoadClassList(const char* fileName):` Load in the log file and extract all the class information to sm_ClassHash and sm_FilterHash.

`ResetClassList():` Reset both class list

`GetClassList()`

`GetMethodList(char*):` Acquire class names and method names from the sm_ClassHash

`FilterExistClass(char*)`

`FilterExistMethod(char*,char*)`

`FilterInsertClass(char*)`

`FilterInsertMethod(char*,char*)`

`FilterDeleteClass(char*)`

`FilterDeleteMethod(char*,char*):` Acquire information from and update to sm_FilterHash

`PrintClass()`

`PrintFilter():` Printing the contents of both lists to standard output for checking purposes

Internals

Two static instances of cl_ClassHash contain the information of all available and selected classes and methods.

`Static cl_ClassHash* sm_ClassHash:` All the available classes and methods

`Static cl_ClassHash* sm_FilterHash:` Selected classes and methods, will be updated dynamically

cl_ClassHash

We decided to implement the list structure used to store the information of classes and methods using hash tables. The availability of Tcl/Tk hash table routines facilitate the maintenance of the lists. Hash table structure provides efficient look up and updates of the list.

Interfaces

`Error Insert(char*,char*)`

`Error GetClass(char***, int*):` Retrieve all the classes, storing the list in an array of char*

`Error GetMethod(char*, char***, int*):` Retrieve all the methods for a certain class

`Error ExistClass(char*)`

`Error ExistMethod(char*,char*):` Check the existence of a class and a method in a certain class.

`Error DeleteClass(char*)`

`Error DeleteMethod(char*,char*):` Delete class or one method of a certain class from the hash table

`Error PrintAll():` Print the whole structure to standard output

Internals

Essentially, cl_ClassHash is structured as a two-dimensional array, with class names in the first dimension and method names in the other dimension. Each dimension is implemented using a hash table. There is a key field and a value field in each element of a hash table. In the hash table for first dimension, class names are stored in the key fields and pointer to the second-dimension hash table stored in the value field. In the second dimension, methods names are stored in the key fields.

Locating a class in the cl_ClassHash involves searching through the first-dimension hash table. Finding a method for a specific class, on the other hand, involves searching through both dimensions.

Two private members exist in cl_ClassHash. These are:

`Tcl_HashTable *tblPtr:` Pointer to the Tcl/Tk hash table object maintained by this class

`Error InsertMethod(Tcl_HashTable* , char*):` Method for inserting a method in the second hash table

Log File

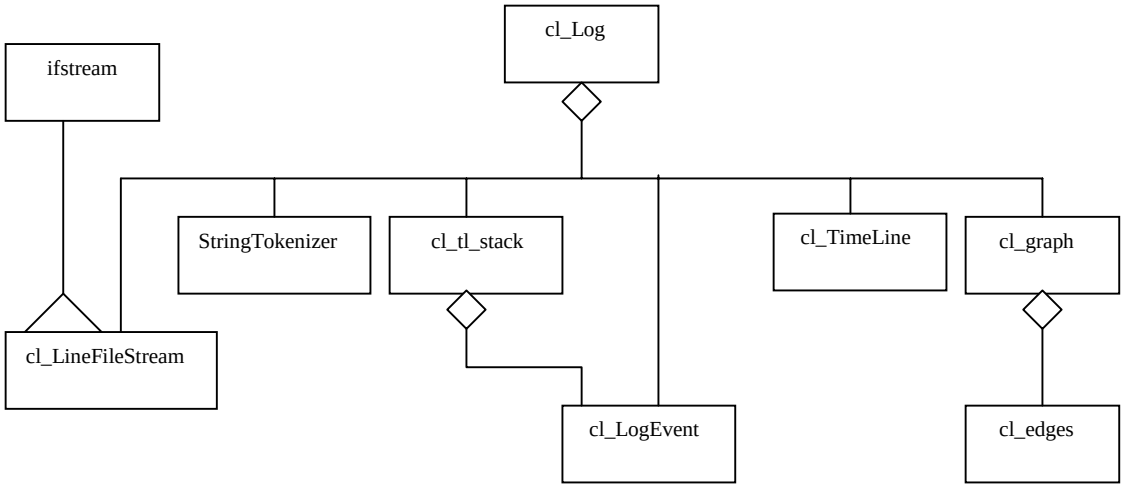
First we will give an overview of the structure of the log-parse subsystem, then we will give detailed descriptions of the interfaces, internal workings, and the limitations/assumptions of the key classes.

The log-file subsystem is responsible for hiding the log-file from the rest of the system. The MSC renderer interfaces with the cl_Log class, asking it for events and time-line information.

Events are the arrows in the MSC diagram and time-lines are the vertical lines.

On instantiation, a cl_Log object will parse the log-file. The cl_tl_stack, StringTokenizer, cl_LineFileStream, and, by inheritance, ifstream, are all used during the parsing. The cl_tl_stack is used to store cl_LogEvents. The cl_StringTokenizer and cl_LineFileStream, condition the log-file input. cl_LineFileStream inherits from ifstream. The cl_graph class, with its cl_edges class members, is used by the cl_Log class in its graph-theoretic heuristic for clustering time-lines for the MSC diagram. The OMT diagram below describes the relationships between the classes.

The methods getNextTimeLine(), getNextEvent() and the booleans hasMoreEvents() and hasMoreTimeLines() are used by the MSC renderer to iterate through the cl_TimeLines and cl_LogEvents respectively.



cl_Log

Interface

`cl_Log(char*):` This constructor takes the char* filename of the log-file as a parameter and parses it, filling the m_logEventArr array with log events and the m_timeLineArr array with time lines.

Here is the structure of a line of the log-file:

```
classname 0x00000000 enter|return timestamp methodname
```

`classname` = the name of the class.

`0x00000000` = the memory address of the object/class instance.

`enter|return` = `enter` on AutoLog allocation, signaling method invocation. `return` on AutoLog deallocation, signaling method return.

`timestamp` = system time at log event.

`methodname` = the name of the method.

The parsing is not a simple task. In order to draw message passing lines from time-lines to time-lines, we must know what we are returning to. We need a stack for this. For example, in the log-file we may see the lines:

```
cl_FullRequestHandler 0x0006b910 ENTER -0090 HndleGetHeadRequest(cl_Request* req)
cl_ConfServer 0x0006b918 ENTER -0090 IsToBeRidirected(const char* src)
cl_ConfServer 0x0006b918 RETURN -0090 IsToBeRidirected(const char* src)
cl_RequestHandler 0x0006b910 ENTER -00008000 ResolvePath(cl_Request* req)
```

After LINE 3 we return to the cl_FullRequestHandler at memory address 0x0006b910. Function call-return is context-free. We keep track of where we are with a cl_tl_stack object – pushing and popping cl_LogEvents as we parse the log file. The cl_tl_stack supports push, pop, and peek.

`Error getNextEvent(cl_LogEvent& leFrom, cl_LogEvent& leTo):` Two cl_LogEvents are passed to this routine by reference. This routine copies over the next two events in the m_logEventArr and moves the iterator forward. Taken together these two events contain the information to draw an “arrow” in the MSC diagram from one time-line to another.

`Error getNextTimeLine(cl_TimeLine&):` This routine is used to iterate through the cl_timeLine objects stored in m_timeLineArr.

`Error getTimeLine(cl_TimeLine&,int):` Alternatively, we can extract a cl_TimeLine object from the m_timeLineArr by specifying an index in to the array.

`int hasMoreTimeLines():` This routine returns 1 if more timeLines remain to be iterated through, 0 otherwise.

`int hasMoreEvents():` This routine returns 1 if more cl_LogEvents remain to be iterated through, 0 otherwise.

`log_state currentState`: The cl_Log constructor does a lot of work – it opens the log file, parses it and inputs all the information into the m_logEventArr and m_timeLineArr arrays.

`void resetLogEventIter()`: The user calls this routine to reset the m_logEventArr iterator.

`void resetTimeLineIter()`: The user calls this routine to reset the m_timeLineArr iterator.

`void printLog(char*)`: The user calls this routine to print out the contents of the log.

Internals

`Error addToEventArr(cl_LogEvent&)`: Add this event to the m_logEventArr. This function adjusts the size of the array if necessary.

`int getTimeLineIndex(char*,char*,char*,int)`: This function is called by the constructor to either create a new cl_TimeLine object, if appropriate, or return the index of the appropriate existing one. If this function creates a new cl_TimeLine then it adds it to the m_timeLineArr array, adjusting the size of the array if necessary.

The routine creates a new cl_TimeLine object in the following cases:

1. On encountering a constructor call. If an object of the same class has existed at this objects memory address before we will have a previous cl_TimeLine with the same (class name, memory address) pair. The field monotonicStamp is used to distinguish such cl_TimeLines. This cl_TimeLine is assigned the incremented monotonicStamp value of the most recent cl_TimeLine sharing the same class name and memory address.

2. On encountering a new (class name, memory address) pair.

`Error orderTimeLinesForMscDiag()`: This routine is called by the constructor to compute the ordering of the cl_TimeLines in the MSC diagram. Over the course of the trace each time-line will send a certain number of messages to each of the other time-lines. We can think of the number of messages as the weight of an edge connecting two time-line vertices. We want to group together those time-lines that communicate with each other frequently – for the benefit of the user trying to make sense out of the diagram. Selecting an optimal such layout is equivalent to a well known problem – selecting the maximal hamiltonian path, or if the graph is not fully connected selecting the maximal hamiltonian paths over the disconnected subgraphs. Unfortunately, this is a very hard problem. So, rather than seeking to globally optimize we use a heuristic approach to select a good approximation to the optimal layout. We now describe our clustering heuristic.

Clustering Heuristic: Select the maximal weight edge from the graph. Delete this edge. If no list exists then create one with these two elements as the first two members of the list. Continue extracting the maximal edge from the graph. Every time we see an edge that has not already been put in to the list then scan the list from head to tail, computing the net impact of the node insertion on the sum of all the edge weights across the linked list. Insert the vertex where it will have the maximal impact. Finding the maximal edge is implemented in $O(n^2)$ time in the getMaximalEdge method of the cl_graph class. Scanning the list for each vertex will take at most $O(n^2)$ time. So in total our approximation runs in $O(n^2)$ time. Our experience with this heuristic has been very successful – our profiles have been well-clustered.

cl_TimeLine & cl_LogEvent

cl_TimeLine Interface

cl_TimeLine objects represent the vertical time-lines of the MSC diagram.

`Error copy(cl_TimeLine&):` copies the values of the referenced object into our own members.
`char* m_className:` the name of the class.
`char* m_memAddr:` the memory address of the object.
`int m_monotonicStamp:` a stamp used to differentiate between objects of the same class that share the same memory address over time.
`int m_locMscDiag:` the physical location in the msc diagram, computed by heuristic to maximize clustering.

cl_LogEvent Interface

Individually, cl_LogEvents represent one end of an MSC message arrow.

`Error copy(cl_LogEvent&):` copies the values of the referenced object into our own members.
`int isDummyEvent():` returns 1 if this is a "dummy" event, 0 otherwise. A dummy event is used to mark events
`int m_timeLineIndx:` the index of the time-line associated with this event.
`char* m_methodName:` the method name of this event.
`char* m_timestamp:` the timestamp.
`int m_direction:` the direction. 0 for method return. 1 for method enter.

Sometimes we need cl_LogEvent placeholders. These are known as "dummy" events. Consider the log:

```
cl_ConfServer 0x0006b918 ENTER 000007000 cl_ConfServer()
cl_ConfServer 0x0006b918 RETURN 000008000 cl_ConfServer()
```

When clients ask the cl_Log for events, we give out the next two in the m_LogEventArr. If the log-file contained only those two lines immediately above then the cl_Log would, on parsing, create a 'dummy' event that we come from on entering cl_ConfServer. So the sequence of log events would be:

```
DUMMY EVENT
cl_ConfServer 0x0006b918 ENTER 000007000 cl_ConfServer()
cl_ConfServer 0x0006b918 RETURN 000008000 cl_ConfServer()
DUMMY EVENT
```

Dummy cl_LogEvents do not belong to a time-line. They are just placeholders. In the MSC diagram we see 'dummies' as interrupted arrows.

cl_StringTokenizer

This class takes a string and breaks it up into a sequence of tokens. The cl_LogFile constructor has a local cl_StringTokenizer object which it uses to parse the log file.

Interface

`StringTokenizer(char* str, char* = 0, char* = 0):` The constructor takes three arguments – the string to be tokenized, the string of delimiters, and the string of delimiters which are to be returned during parsing.

`int countTokens(): The number of tokens in the string.`

`int hasMoreTokens(): Returns 1 if the class has more tokens, 0 otherwise.`

`char* nextElement(char* = 0): Returns a pointer to the next element.`

`char* restOfString(): Returns the rest of the string.`

Internals

`char* m_Delims:` This string contains all the delimiters. Delimiters are characters that cannot be members of tokens.

`char* m_RtrnDelims:` This string contains all the delimiters that are to be returned when encountered on calls to nextElement.

`char* m_Str:` The string under analysis.

`int m_StrLen:` The length of m_Str.

`int m_StrOffset:` Our current position in m_Str - a 0-based index.

`int inDelims(char):` Returns 1 if the char parameter is in the m_Delims string, 0 otherwise.

`int inRtrnDelims(char):` Returns 1 if the char parameter is in the m_RtrnDelims string.

cl_LineFileStream

This class inherits from ifstream, adding the method getLine() for reading a file line-by-line.

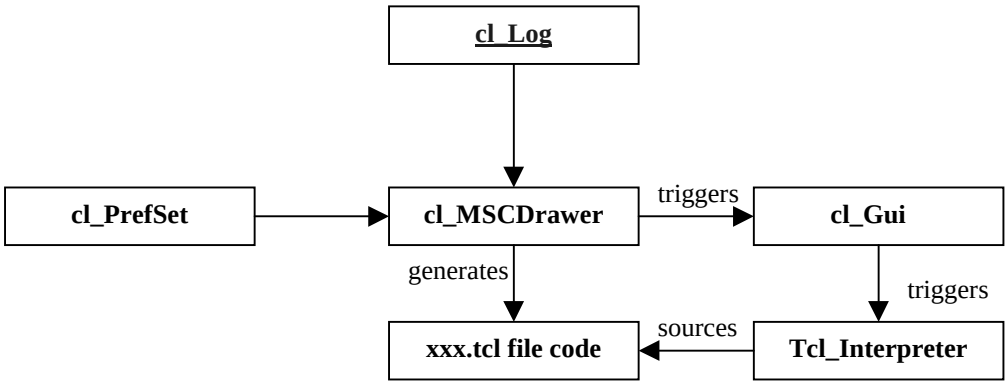
Interface

```
cl_LineFileStream(char* filename) : ifstream(filename) {}
char* getLine();
```

Rendering Engine: cl_MSCDrawer

The rendering engine will be implemented into the cl_MSCDrawer class.

The cl_MSCDrawer class is responsible for translating a cl_Log structure into an MSC diagram that can be plotted on a Tcl/Tk canvas. The idea behind is based on the possibility offered by Tcl/Tk to "source" code at running time. The Tcl interpreter lets you therefore generate code that will be dynamically loaded at running time, adding capabilities to your program. The approach taken is to exploit this possibility and generate the Tcl code necessary to plot the MSC corresponding to the given cl_Log structure. Once the code has been generated, cl_MSCDrawer will trigger an event in the GUI, which will then show the new MSC to the user by loading it into the Canvas widget. The diagram below shows the information flow among the major components involved into the process.



The first step of the process is performed by the cl_Log class, which parses the log file and extracts a sequence of events from it. While iterating through this information repository with cl_Log's getNextEvent(), cl_MSCDrawer generates the appropriate Tcl code into a new file. The code consists of a sequence of Tcl commands that use a Canvas widget to perform the actual drawing. A set of abstraction routines are pre-programmed in Tcl to reduce the amount of the code to be generated. The latter also provide a level of abstraction over the verbose Tcl/Tk syntax necessary to actually get the drawing. During the rendering operation, cl_MSCDrawer takes into consideration the current user preferences for basic element properties as background and/or foreground colors and for the filtering of classes and methods which the user want to hide in the current rendering.

Such a widget lets the developer assign an area of a window as a canvas; conventional and simple commands such as **`createline`** and **`createbox`** will be used to the do the actual depicting.

We use the canvas widget and its command syntax to plot the final MSC. However, the abstraction offered by these procedures can be seen as a high-level drawing language customized for our needs. This use of such a language ensure that our rendering will not be bound to Tcl. In the future, we could use alternative renderers, interpreting this drawing language appropriately.

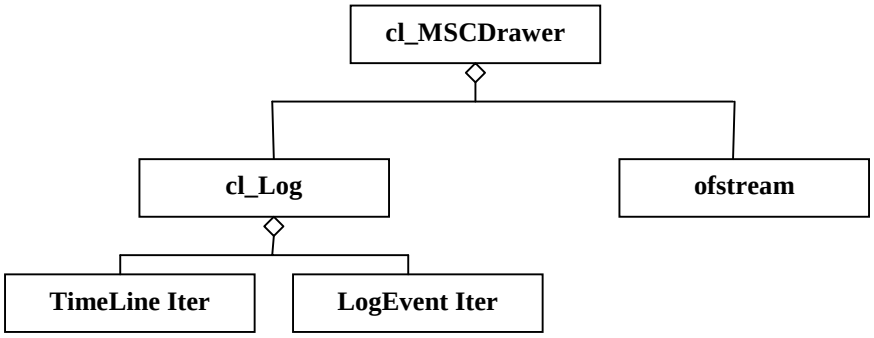
An example of the generated code follows:

```
ResetCanvas
DrawInstance cl_RequestHandler 0x0006b910
DrawInstance cl_FullRequestHandler 0x0006b910
DrawInstance cl_Server 0x0006b908
DrawInstance cl_ConfServer 0x0006b918
DrawInstance cl_RequestHandler 0x0006b914
DrawIn 0x0006b910 cl_FullRequestHandler "cl_FullRequestHandler(cl_Server* server)"
DrawOut 0x0006b910 cl_FullRequestHandler "cl_FullRequestHandler(cl_Server* server)"
```

```
DrawIn 0x0006b914 cl_RequestHandler "cl_RequestHandler(cl_Server* theServer)"
DrawOut 0x0006b914 cl_RequestHandler "cl_RequestHandler(cl_Server* theServer)"
DrawIn 0x0006b918 cl_ConfServer "cl_ConfServer()"
DrawOut 0x0006b918 cl_ConfServer "cl_ConfServer()"
DrawIn 0x0006b908 cl_Server "cl_Server(const char* prefFile)"
DrawMethod 0x0006b908 cl_Server 0x0006b918 cl_ConfServer "LoadPrefs(const char*
prefsFile)" 1 1
DrawMethod 0x0006b918 cl_ConfServer 0x0006b908 cl_Server "cl_Server(const char*
prefFile)" 0 1
DrawMethod 0x0006b908 cl_Server 0x0006b918 cl_ConfServer "PrintStatus(ostrstream &ostr)"
1 1
DrawMethod 0x0006b918 cl_ConfServer 0x0006b908 cl_Server "cl_Server(const char*
prefFile)" 0 1
```

We identify five commands:

1. **`ResetCanvas`**: clears the canvas and reinitialize the local Tcl variables for a new drawing.
2. **`DrawInstance`**: creates a new timeline starting from the class name and the object memory address pair. The representation will be a box on the top of the canvas and a vertical line
3. **`DrawIn`**: displays a trace representing the entering of a method. The actual sender of such a message is unknown in this case. This might be due to the fact that we did not trace the sender (no auto-log class has been inserted).
4. **`DrawOut`**: displays a trace representing the exiting of a method. The actual class we are returning to is unknown in this case. This might be due to the fact that we did not trace the receiver (no auto-log class has been inserted).
5. **`DrawMethod`**: displays the actual delivering of a message from one instance to another one. Both instances are identified by the parameter list. The same command can be used to express the returning from a method call: the last two parameters are used to discriminate between the two cases.



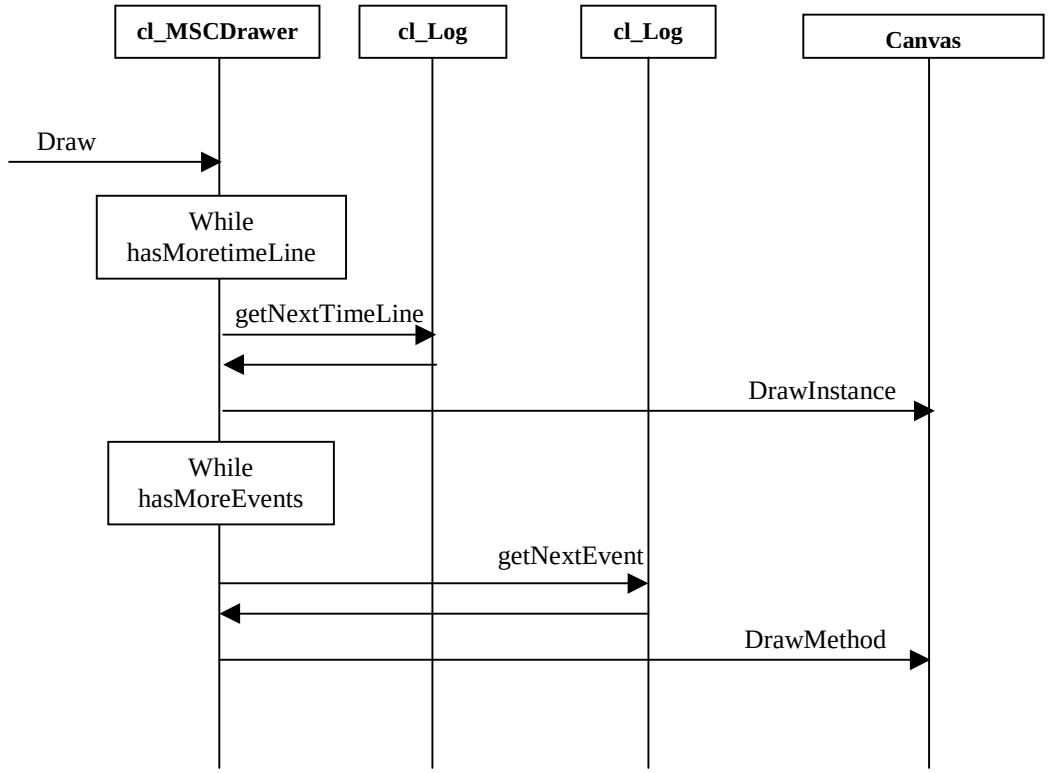
Internals

The internal data structures of cl_MSCDrawer are exposed in the above OMT class diagram. Every cl_MSCDrawer will accesses an instance a cl_Log object. The data stored in such an object will be then rendered into the MSC diagram. The cl_Log is queried repeatedly for events – which are transformed into new Tcl commands, which are saved into the Tcl/Tk generated file. This file is sourced at run-time, causing the refresh of the MSC canvas.

cl_MSCDrawer inherits from the cl_Gui class: this is motivated by the fact the cl_MSCDrawer interacts with the GUI. The parsing of the log file can fail, in which case the user must be notified. This is done through the cl_GUI::ErrorDialog method.

Interfaces

The following MSC (yes, we use them in our design too!) shows a typical execution flow during the rendering of a new MSC diagram. The **`Draw(logFileName)`** method triggers the interpreting of the supplied structure and generation of the corresponding Tcl drawing code.



Limitations and Assumption

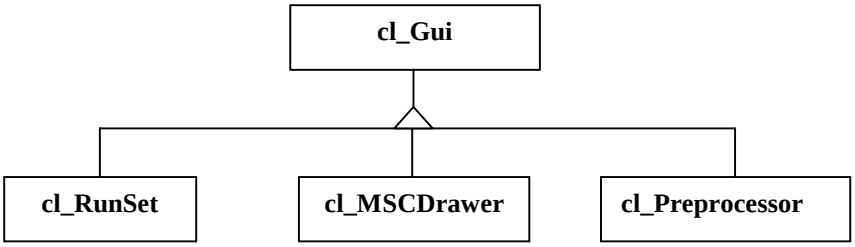
Currently, the criteria used for identifying timelines in a canvas uses a (className, memAddress) pair. This can be extended in the future in order to take advantage of the time stamp associated with each timeline during the parsing of the log file.

cl_Gui

cl_Gui is the class offering the hook from the Tcl/Tk code to the C++ core of VOTE. It provides a set of methods abstracting on the communication between the C++ code and Tcl/Tk. Defining such a class let us group the common implementation of all those classes which need to send feedback to the Tcl/Tk gui.

Internals

The implementation of cl_Gui is currently a set of static methods which embed the syntax of the communication between Tcl/Tk and C++. The following inheritance diagram shows the relatioships of cl_Gui with other components of the system.



All the three derived classes above have to interact with the Tcl. The way this is achieved is common for all of them. No matter how many new Tcl commands each of these classes extend from the basic Tcl, all those commands will be addressed to a single method in their code. Such a method will then dispatch accordingly to the first argument passed (command name) to the appropriate method into the class. If necessary, the dispatcher will also take care of the marshalling of the command by type casting all the parameters of the handling method.

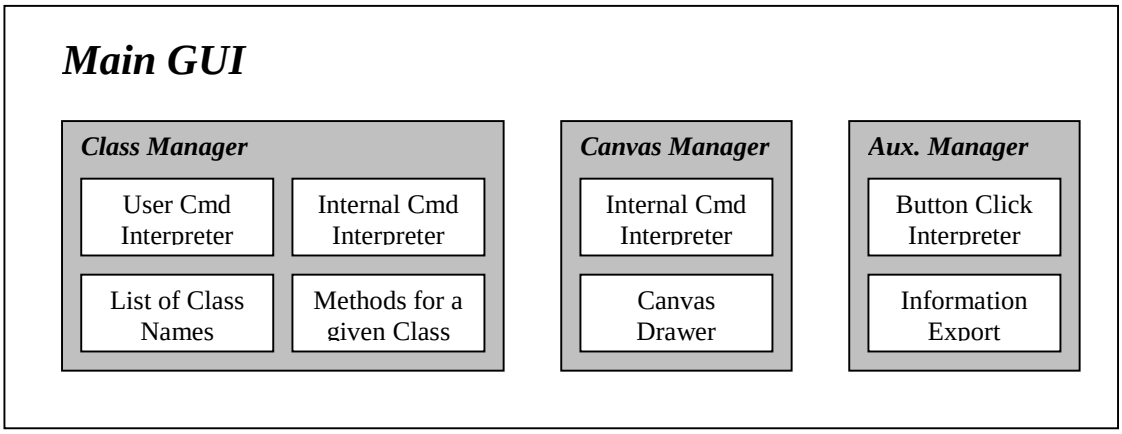
As an example, cl_RunSet extends the Tcl basic language by adding seven new commands. A single dispatcher will be invoked whenever one of those commands is executed in Tcl. The dispatcher will then extract the correct number of parameters and call the method supposed to handle the event. The parameters are then returned back to Tcl through the interfaces offered by cl_Gui.

Interfaces

cl_Gui offers interfaces to reset the parameter list returned to Tcl, and methods to actually add a new string into such a list. A particular convention has been used into the implementation for the error management and notification. The first parameter returned to Tcl from a command implemented into C++ is an error code. The client Tcl code is responsible for checking for an “OK” value on such a parameter before going on with the normal execution. If an error is detected, than a error dialog box has to be shown to the user and the execution should take a different path.

All this is achieved by making use of the cl_Gui::ErrorDialog method.

Subsystem Diagram: Main GUI



The Main GUI

The main GUI will be primarily implemented in Tcl/Tk scripts. It communicates with other parts of the Tcl script for most of its inputs. Of course it also must possess the ability to interpret user commands. We focus on three main windows: The canvas on which the MSC is drawn, the list box where the class names appear and the list box where the methods appear. With the other auxiliary control buttons, we also need an auxiliary manager to help smooth the communications. Consequently, we have three sub-modules, the Class Manager, the Canvas Manager, and the Auxiliary Manager.

Class Manager

The Class Manager juggles inputs from the user as well as other procedures of the Tcl script to maintain the two list boxes containing the class names and their methods. The manager that handles user inputs will automatically update the methods list box when a class name is selected. The other manager who receives internal commands mainly take care of updating the list of classes, usually following some user file selection.

The implementation of the Class Manager is stored into two modules: **`vote.tcl`**, where the implementation of the GUI widgets is defined and which is interpreted at boot time, and in **`handling.tcl`**, which implements the necessary procedures to populate pop-up menus and listboxes following from a user action.

Canvas Manager

The Canvas Manager is responsible for handling the updates of the canvas. It is closely bound to the C++ module that handles the generation of Tcl script that the canvas sources. An interpreter lets this manager know that the canvas is to be updated so that it can perform the necessary operations.

The **`CanvasLib.tcl`** collects all the necessary procedures to reset and clean a canvas and perform all the drawing operations. The implementation of **`DrawInstance`** and **`DrawMethod`**, the commands generated from the cl_MSCDrawer are kept in this module.

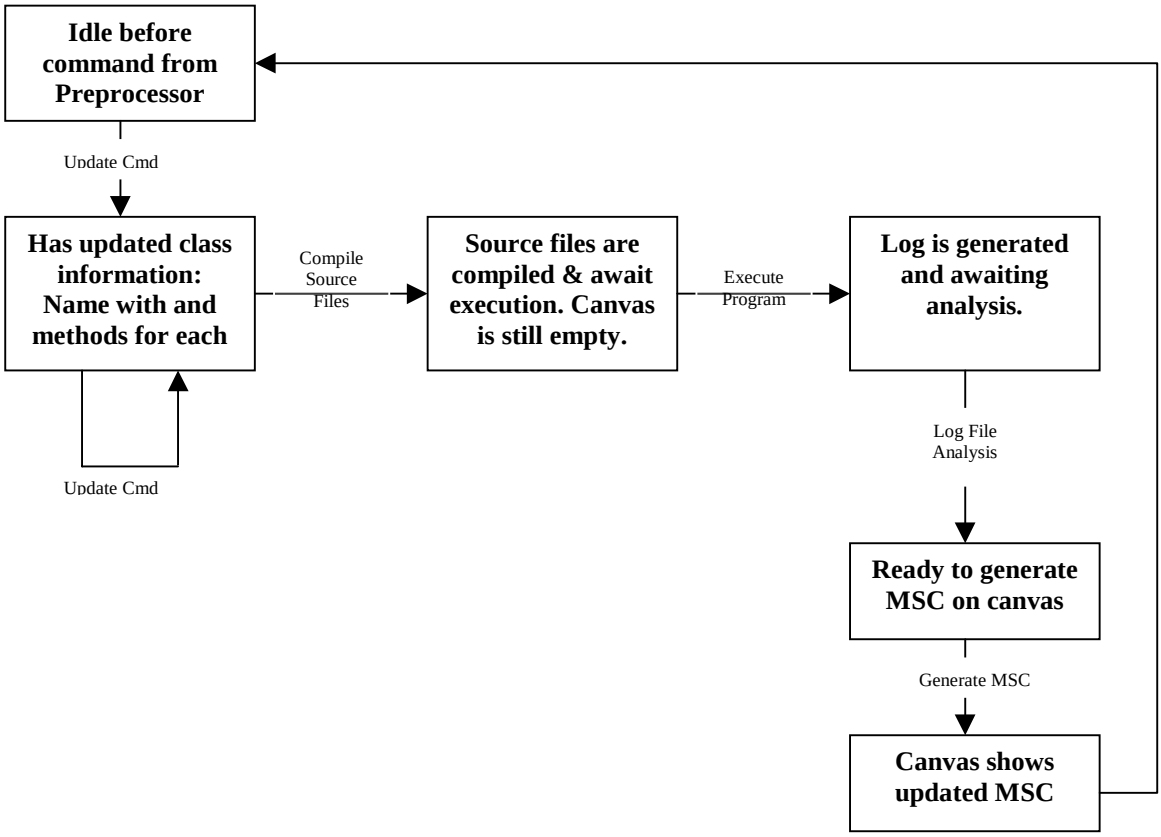
Auxiliary Manager

There are buttons called Start, Stop and Run, which are the controls for other modules as well. There needs to be, naturally, an interpreter that translates each of the commands, internal and external for the GUI. Conversely, information that GUI holds may be useful for other modules; so we need to have a part that exports information outwards.

Most of these auxiliary widgets are collected into the menu bar and the drawing tool bar. This includes both buttons, which let the user select the starting and ending method for a new drawing, and the draw and reset button. The actual declaration of those widgets is stored into the **`vote.tcl`** module.

Additional modules and dialog boxes are defined for handling the user preferences and the run-time settings. Those are collected into the **`Prefs.tcl`**, **`build.tcl`**, and **`select.tcl`**.

Main GUI Flow Chart



Order of Events in the Main GUI

This isolated flow chart reveals the order of updating the three main windows of the GUI: The canvas where the MSC goes, the list box for the class names and the list box for the methods. The windows remain inactive until outside sources trigger change. These changes include a "Update" command from the Preprocessor, a "Compile" command from the user, which may or may not be coupled with the "Execute" command, and finally the "Generate MSC" command.

With the "Update" command the two windows containing the class name and associated method names will be updated accordingly. When a user clicks on a class name to select it, the windows with method names should change its contents to reflect the methods of the current selected class. This can be done easily with simple procedures and the hash table mentioned in the documentation of the Preprocessor.

The flow is similar to that of the Preprocessor until the point where the log file analyzer comes in. The termination of the analysis will trigger the updating of the Canvas window by "sourcing" another Tcl file generated during analysis. After updating the MSC, we're back to the initial state awaiting further requests from the user.

Limitations and Assumptions

As any other project is subject to the triple constraint, VOTE contains many limitations in the form of assumptions made as well as performance metrics. As we have done most of our development on Windows NT, the program needs to be modified before running on Unix. This is a higher level limitation of VOTE. The following is a list of these limitations per module, as they will be fixed/enhanced in the next version of VOTE.

- The preprocessor currently assumes that the files that the user specifies contain method implementations and not only class declarations. It only searches for the actual body of the function and insert the AutoLog code there. In other words, it does not handle in-line functions that may appear in a class declaration.
- During source file modification, the preprocessor makes a backup of the file appending the extension ".backup" after the original file name. The way it works now, the preprocessor won't operate on a file if its backup already exists, thus disallowing double processing of files. This leads to two consequences when a user wishes to change his code. If the change that the user wants to perform does not involve additions of functions, then the user can operate on the modified file and everything will still work fine. However, if the change involves adding one or more functions, the user needs to copy the backup file to the original file, start editing from there and modify that file again by selecting the *Modify Sources*. With this way of preprocessing, one assumption that we have is that if a user wants to modify a source file of name foo.cpp, he must make sure that he has no files with the name foo.cpp.backup in the same directory. Otherwise, that file will not be changed.
- If the user exits the program using any other way than selection from the *Quit* command button of the *File* menu, the, the changed files won't be restored.
- During the building and running phases of the preprocessing stage, the preprocessor takes in user preferences from the Tcl script provided by the PrefSet module. This automatically implies that the two have to be done via the command line. Our assumption that the user will provide a makefile for the building is not always justified; but it does simplify things a lot.
- We assume that the user is working in a directory where he has read and write access. Numerous system calls are invoked in the Tcl script and they assume the user can create, copy, and delete files in the directories he specified.
- Other small miscellaneous assumptions we made include: the compiler and linker commands that the user specify must be a single command (i.e. there cannot be more than one command given, even separated by semicolons); and the directories given in the preferences dialog should be existent
- The log file name should not conflict with any existing file in the working directory. VOTE will overwrite any file with the same name.

Testing

Our software architecture consists of several subsystems – the GUI, the preprocessor substitution engine, the log-file parser, the MSC renderer, and the configuration interpreter. The interfaces between subsystems – whether a standard file format or set of public methods, were well-defined and agreed upon before implementation. This allowed the subsystems to be developed in parallel by different designers. The subsystems were tested in isolation before the merge. This allowed the merge to proceed very smoothly, with few problems. Considering that the system stands at over 7500 lines of code, the fluid merge was a real accomplishment.

Individual Subsystems.

What strategies did we use to test the subsystems?

Early on, we drafted documents specifying the interfaces between components. Master copies were made public on one of our subdirectories. These docs were referred to frequently as we designed test cases for the individual subsystems.

Each of us did both white-box and black-box testing on our subsystems. The white-box testing involved stepping through code and watching execution unfold, keeping an eye out for strange or unexpected behavior.

With black-box we specified an input, executed the system and checked that any output and the final state of the object were correct. For most VOTE subsystems the input space is very large. Just take a look at the preprocessing engine – it is designed to take most C++ programs as input. With such a vast test space, we had to be very smart - selecting test cases that seemed most likely to present problems. For example, with the cl_Log we made sure that we tested the parsing with input sequences that reached the bottom of the stack frequently – forcing the generation of "dummy" events. As well, knowing that the cl_Log objects expand the size of their m_logEventArr and m_timeLineArr arrays exponentially when they fill up, we made sure to force exponential expansion with large log-files with many time-lines.

Once we had performed in-depth smart testing, corrected the revealed bugs, and were quite confident in the correctness of our subsystems we began the merge.

Merged System

We reached a stable state with the merged system much faster than with the individual components. Most of the problems that we encountered during merge were superficial and easily fixed. We reused many of the test log-files from the subsystem testing phase, concentrating on studying the correctness of the MSC diagram, comparing it with the original log-file. This technique worked quite well. Early on, we discovered that some events were not properly represented in the MSC diagram. We corrected this quickly by parsing the log-file with a stack and generating "dummy" events when we leave and re-enter the top-level of execution. After this we had few problems and we feel very confident in the correctness of our final system.